

An Introduction to Neural Networks in AI and Machine Learning

Ollie Darwin
AMP Individual Project

January 25, 2026

Abstract

This report aims to explore the fundamentals of neural networks within the broader context of artificial intelligence (AI) and machine learning (ML). Starting with the basic computational unit—the perceptron—we work through activation and loss functions before looking at how networks are actually structured and trained. We then apply these ideas to a simple implementation on the MNIST dataset of handwritten digits, and finally consider some extensions (like convolutional and recurrent networks) as well as the limitations of the whole approach. The goal is to understand not just *how* neural networks work, but to develop some intuition for *why* they work the way they do.

Contents

1	Introduction	3
1.1	Background and Motivation	3
1.2	Scope of the Project	3
2	Perceptrons: The Basic Building Block	4
2.1	History of the Perceptron	4
2.2	Mathematical Formulation	4
2.3	Limitations of Single-Layer Perceptrons	5
3	Sigmoid Neurons and Activation Functions	5
3.1	From Threshold to Sigmoid	5
3.2	Choosing Activation Functions — Pros & Cons	7
4	Loss Functions and Learning Objectives	8
4.1	What is a Loss Function?	8
4.2	Typical Loss Functions for Classification and Regression	9
4.3	Gradient Descent and Optimisation Overview	10
5	Structure of Neural Networks	11
5.1	Feedforward Networks and Multi-Layer Perceptrons	11
5.2	Hidden Layers, Depth vs Width	12
5.3	Other Architectural Considerations (Dropout, Regularisation)	12

6	Back-Propagation and Training Neural Networks	14
6.1	Forward Pass and Loss Computation	14
6.2	Back-Propagation Algorithm — Derivation and Intuition	14
6.3	Practical Considerations: Learning Rate, Vanishing/Exploding Gradients, Weight Initialisation	16
7	Application: Implementing a Simple Neural Network for MNIST	17
7.1	Dataset Description	17
7.2	Network Architecture Used	18
7.3	Training Procedure and Hyperparameters	18
7.4	Results and Discussion	19
7.5	Code Snippets / Implementation Details	20
8	Extensions: Convolutional and Recurrent Neural Networks	21
8.1	Convolutional Neural Networks (CNNs) — Motivation and Basics	22
8.2	Recurrent Neural Networks (RNNs) — Motivation and Basics	23
8.3	Where They Fit and When to Use Them	25
9	Limitations and Challenges of Neural Networks	25
9.1	Overfitting, Interpretability, Data Requirements	25
9.2	Computational Costs and Scalability	26
9.3	Current Research Challenges (e.g., generalisation, adversarial examples) .	27
10	Conclusion	28
10.1	Summary of Key Insights	28
10.2	Possible Future Work	29

1 Introduction

1.1 Background and Motivation

When you first hear about neural networks, there's something almost magical about the idea that a computer can learn to recognise faces, understand speech, or even generate convincing text. It's quite easy to think there must be some unexplainable force at work. As artificial intelligence (AI) and machine learning (ML) have become perhaps the most rapidly developing fields in modern computing, it seemed like a worthwhile challenge to try to understand what's actually going on underneath all the hype.

At their core, many of the recent breakthroughs in AI—things like image recognition, natural language processing, and generative models—are powered by what are called *artificial neural networks* (ANNs). These are computational models that take their inspiration from the structure and function of the human brain, although it's worth noting that the analogy is quite loose. They consist of layers of interconnected nodes (often called *neurons*) that process input data and, through a process of mathematical optimisation, learn to recognise patterns.

What I found particularly interesting when researching this topic is that neural networks bring together concepts from linear algebra, calculus, and probability in order to model complex, non-linear relationships between inputs and outputs. Despite their apparent complexity, even the most advanced networks are really built from quite simple components—things like the perceptron, activation functions, and loss functions—that together form what has become known as deep learning. Understanding these building blocks seems like a good place to start.

1.2 Scope of the Project

The aim of this project is to explore the mathematical principles underlying neural networks and to demonstrate how they can be implemented to perform a fairly simple image-classification task. More specifically, the project will:

- Introduce the perceptron and sigmoid neuron as the fundamental computational units of a neural network.
- Explain key mathematical components such as activation and loss functions.
- Describe how neural networks are structured and trained using back-propagation algorithms.
- Apply these ideas to construct and train a simple neural network on the MNIST dataset of handwritten digits.
- Discuss some of the limitations of this basic approach and briefly look at extensions such as convolutional and recurrent neural networks.

Overall, this report focuses on the mathematical and computational aspects of neural networks, rather than their software engineering or hardware implementations. I've tried to include derivations of key equations—such as those governing gradient descent and back-propagation—because working through these really helped clarify for me how networks actually learn from data. While the example implementation uses Python, the

emphasis is meant to be on the underlying theory rather than the code itself. The discussion concludes by reflecting on some of the challenges of training neural networks and possible directions for future exploration.

2 Perceptrons: The Basic Building Block

2.1 History of the Perceptron

The perceptron was introduced by Frank Rosenblatt[17] in 1958, making it one of the earliest computational models of learning that drew inspiration from biological neurons. The basic idea was to classify inputs by computing a weighted sum and applying a threshold function—essentially mimicking a very simplified version of how a biological neuron might fire based on its inputs. Rosenblatt’s work was a significant milestone in the history of artificial intelligence and led to a lot of early optimism about what neural computation might achieve.

However, in 1969, Marvin Minsky and Seymour Papert published *Perceptrons: An Introduction to Computational Geometry*[14], which was a mathematical analysis that revealed some quite serious limitations of single-layer perceptrons. They showed that such models could only represent linearly separable functions, and famously demonstrated that they fail on tasks like the exclusive-OR (XOR) problem. This finding initially slowed progress in neural network research considerably—some sources describe this period as an “AI winter” for neural networks. It was only later that the development of multi-layer networks and the back-propagation algorithm[18] brought the field back to life and formed the foundation of what we now call deep learning.

2.2 Mathematical Formulation

A perceptron takes several binary inputs, $x_1, x_2, \dots$, and produces a single binary output:

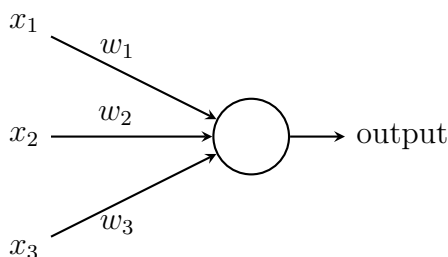


Figure 1: A simple perceptron with three input features.

In the example shown above, the perceptron has three inputs: x_1, x_2, x_3 . In general, a perceptron can have any number of inputs. Rosenblatt proposed a fairly simple rule to compute the output. He introduced *weights*, $w_1, w_2, \dots$, which are real numbers that determine how much influence each input has on the output. The neuron’s output, either 1 or 0, is then determined by whether the weighted sum $\sum_j w_j x_j$ is greater than or less than some *threshold value*. Just like the weights, the threshold is a real number and acts as a parameter of the neuron. Putting this more precisely:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq \text{threshold}, \\ 1, & \text{if } \sum_j w_j x_j > \text{threshold}. \end{cases}$$

This is essentially all there is to the basic perceptron. It’s worth pausing to appreciate how simple this is—it’s really just a weighted sum followed by a comparison. The “learning” part comes from adjusting the weights and threshold based on examples, which we’ll come back to later.

2.3 Limitations of Single-Layer Perceptrons

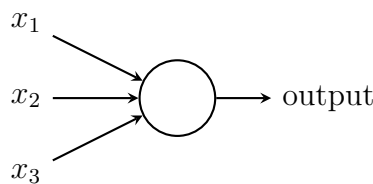
Although the perceptron introduced the fundamental idea of learning by adjusting weights, its capabilities turn out to be severely limited. A single-layer perceptron can only represent what are called linearly separable functions—meaning it can only correctly classify data points that can be divided by a straight line (or, in higher dimensions, a hyper-plane). However, many real-world problems, such as the XOR function, are not linearly separable and therefore cannot be solved by this simple model.

This limitation, which Minsky and Papert (1969) made very clear in their analysis, demonstrated that single-layer architectures simply don’t have enough representational power for more complex decision boundaries. It’s a bit like trying to separate two interleaved spirals with a single straight line—no matter how you position the line, it just won’t work. This is fundamentally what motivated the development of multi-layer networks and non-linear activation functions, which we’ll look at in the following sections.

3 Sigmoid Neurons and Activation Functions

3.1 From Threshold to Sigmoid

Now that we’ve had a look at the perceptron and how it works, we can move on to the *sigmoid* neuron. We can draw a sigmoid neuron in the same way we drew perceptrons:



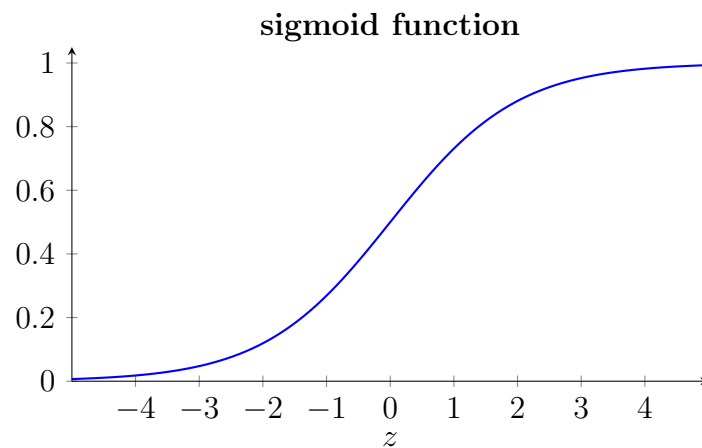
Like the perceptron, sigmoid neurons take inputs $x_1, x_2, \dots$. However, instead of being restricted to just 0 or 1, these inputs can take any value *between* 0 and 1. Also like a perceptron, a sigmoid neuron has weights for each input, $w_1, w_2, \dots$, and an overall bias, b . The output is $\sigma(w \cdot x + b)$ where σ is what’s called the sigmoid function, defined by:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

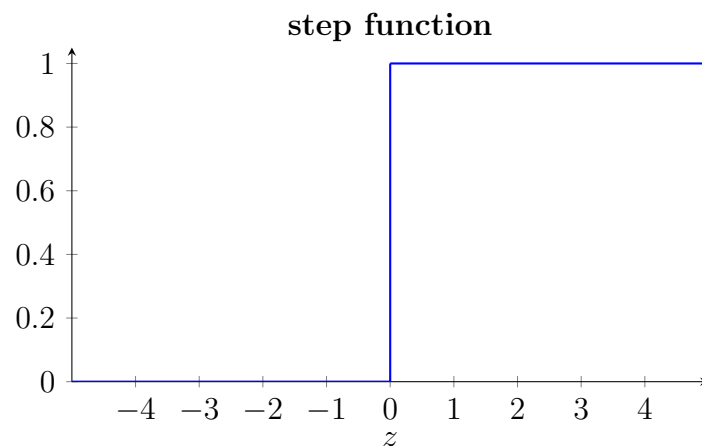
To write this out more explicitly, the output of a sigmoid neuron with inputs $x_1, x_2, \dots$, and weights $w_1, w_2, \dots$, with bias b is:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

At first glance this might look quite different from the perceptron, but if we look at the shape of the sigmoid function it starts to make more sense. Below is a plot of the sigmoid function:



This shape is essentially a smoothed out version of a step function:



If σ were the step function instead, then the sigmoid neuron would *be* a perceptron, since the output would be 1 or 0 depending on whether $w \cdot x + b$ was positive or negative. The actual σ function gives us a smoothed version of this behaviour. The key advantage of

this smoothness—and this is quite important for what comes later—is that the sigmoid function is differentiable everywhere. This means we can calculate gradients, which is essential for the learning algorithms we’ll discuss.

3.2 Choosing Activation Functions — Pros & Cons

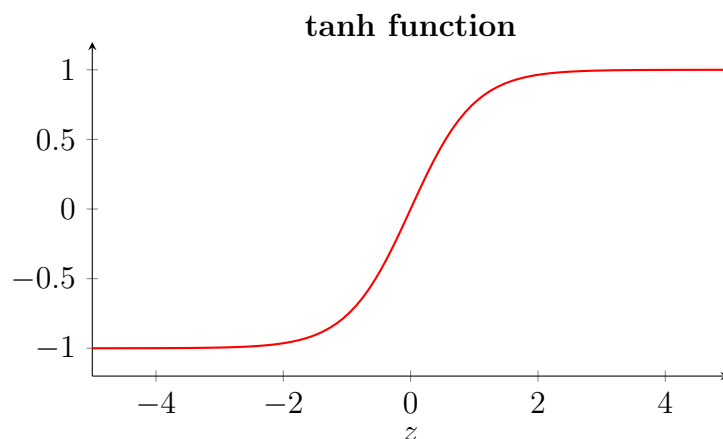
While the sigmoid function was historically important in neural network development, modern architectures tend to use a variety of different activation functions. Each has its own characteristics that make it more or less suitable for different situations. Understanding why certain activation functions work better than others took me a while to get my head around, but it’s actually quite important for building effective networks.

Hyperbolic Tangent (Tanh)

The hyperbolic tangent function is defined as:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1}$$

Unlike the sigmoid function, which outputs values in $(0, 1)$, the tanh function outputs values in $(-1, 1)$, which means it’s zero-centered. This property apparently often leads to faster convergence during training, because the gradients are better balanced around zero. I found this argument a bit hand-wavy at first, but it does seem to be borne out in practice.

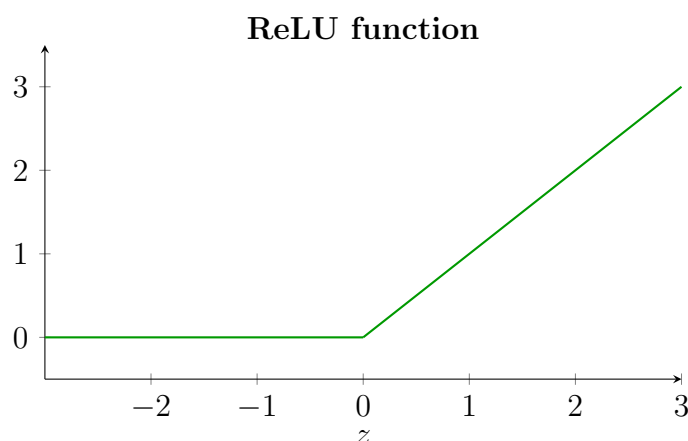


Rectified Linear Unit (ReLU)

The Rectified Linear Unit[15] has become probably the most widely used activation function in modern deep learning. It’s defined quite simply as:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} 0, & \text{if } z < 0, \\ z, & \text{if } z \geq 0. \end{cases}$$

Despite its simplicity—or maybe because of it—ReLU has several advantages: it’s computationally efficient, it helps mitigate the vanishing gradient problem (since its gradient is 1 for positive inputs), and it introduces sparsity in the network (neurons with negative inputs output zero, so they’re essentially “off”). However, there is a downside: ReLU neurons can sometimes “die” during training. If a neuron’s weights are updated such that it always receives negative inputs, it will output zero forever and stop learning. This seemed surprising to me when I first read about it, but it’s apparently a real issue in practice.



Comparative Summary

The table below summarises some key properties of these common activation functions:

Function	Range	Zero-centered	Key Advantage
Sigmoid	$(0, 1)$	No	Smooth, interpretable as probability
Tanh	$(-1, 1)$	Yes	Better gradient flow than sigmoid
ReLU	$[0, \infty)$	No	Fast computation, less vanishing gradient

Table 1: Comparison of common activation functions.

In practice, ReLU and its variants (such as Leaky ReLU and Parametric ReLU) seem to be the default choice for hidden layers in feedforward and convolutional networks these days. The sigmoid and tanh functions are still used in specific contexts—sigmoid for binary classification output layers, and tanh in recurrent neural networks where zero-centered outputs are helpful.

4 Loss Functions and Learning Objectives

4.1 What is a Loss Function?

A neural network learns by adjusting its weights and biases to minimize what’s called a *loss function* (sometimes also called a *cost function* or *objective function*). The loss function essentially quantifies how far off the network’s predictions are from the true labels in the training data. If we have a training set of n examples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(n)}, y^{(n)})\}$,

where $x^{(i)}$ is an input and $y^{(i)}$ is the corresponding target output, the loss function L measures the discrepancy between what the network predicts, $\hat{y}^{(i)}$, and what the true value actually is, $y^{(i)}$.

The goal of training is then to find the parameters (weights w and biases b) that minimize the average loss over the whole training set:

$$\mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^n L(\hat{y}^{(i)}, y^{(i)})$$

Different tasks require different loss functions. What you choose depends on what kind of problem you're dealing with—classification, regression, or something else.

4.2 Typical Loss Functions for Classification and Regression

Mean Squared Error (MSE)

For regression problems, where the output is a continuous value, the mean squared error is commonly used:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2$$

This loss function penalizes large errors more heavily because of the squaring operation. It's also differentiable everywhere, which makes it convenient for gradient-based optimization—you can always compute which way to adjust the parameters.

Cross-Entropy Loss

For classification problems, particularly binary classification, the *cross-entropy loss* (also called *log loss*) is the standard choice. For a binary classification task with true label $y \in \{0, 1\}$ and predicted probability $\hat{y} \in (0, 1)$, the cross-entropy loss is:

$$L(\hat{y}, y) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

This loss function comes from principles of information theory and maximum likelihood estimation—though I won't go into those derivations here. What's important to understand is that it heavily penalizes confident but incorrect predictions, which encourages the network to output well-calibrated probabilities rather than being overconfident.

For multi-class classification with K classes, this generalizes to the *categorical cross-entropy*:

$$L(\hat{y}, y) = - \sum_{k=1}^K y_k \log(\hat{y}_k)$$

where y is a one-hot encoded vector representing the true class, and $\hat{y}$ is the predicted probability distribution over classes (typically produced by a softmax output layer, which we'll see later).

4.3 Gradient Descent and Optimisation Overview

Once we've defined a loss function, the challenge is to actually find the weights and biases that minimize it. *Gradient descent* is the fundamental optimization algorithm used to train neural networks. The idea is conceptually quite simple: compute the gradient of the loss function with respect to each parameter, then adjust the parameters in the direction that reduces the loss.

Mathematically, for a parameter θ (representing any weight or bias), the update rule is:

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

where η is the *learning rate*, a hyperparameter that controls how big each step is. The partial derivative $\frac{\partial \mathcal{L}}{\partial \theta}$ tells us how the loss changes when we tweak θ slightly. By moving in the negative gradient direction, we're moving "downhill" on the loss surface.

Variants of Gradient Descent

In practice, computing the gradient over the entire training set (what's called batch gradient descent) can be computationally expensive, especially with large datasets. Two common variants help with this:

- **Stochastic Gradient Descent (SGD):** Update parameters using the gradient computed from a single training example at a time. This introduces noise into the updates, but allows for much faster iterations.
- **Mini-batch Gradient Descent:** A compromise between the two—compute gradients over small batches of examples (e.g., 32 or 64 samples). This balances computational efficiency with gradient accuracy and is the most commonly used approach in practice.

There are also more sophisticated optimizers like Adam[12], RMSprop[19], and AdaGrad[4] that extend basic gradient descent by adaptively adjusting learning rates for each parameter. These often lead to faster convergence and better performance, though understanding exactly *why* they work better is still an area of active research from what I can tell.

5 Structure of Neural Networks

5.1 Feedforward Networks and Multi-Layer Perceptrons

A *feedforward neural network*, also known as a *multi-layer perceptron* (MLP), is the simplest type of neural network architecture. In a feedforward network, information flows in one direction—from the input layer through one or more hidden layers to the output layer—without any cycles or loops. This is in contrast to recurrent architectures, which we’ll touch on later, where connections can form directed cycles.

A typical feedforward network consists of:

- **Input layer:** Receives the raw input features $x_1, x_2, \dots, x_d$, where d is the dimensionality of the input.
- **Hidden layers:** One or more intermediate layers that transform the input through learned representations. Each neuron in a hidden layer computes a weighted sum of its inputs, adds a bias, and applies an activation function.
- **Output layer:** Produces the final prediction. For regression, this might be a single neuron with a linear activation. For binary classification, a single neuron with sigmoid activation. For multi-class classification, multiple neurons with softmax activation.

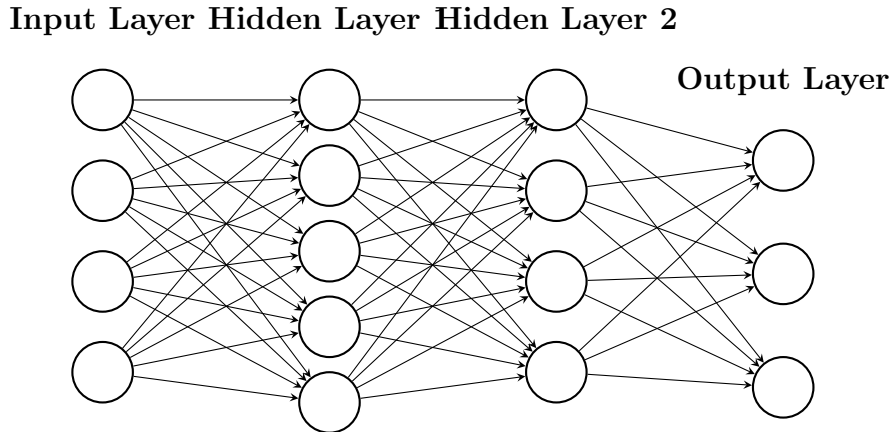


Figure 2: Architecture of a feedforward neural network with two hidden layers. Each connection represents a learnable weight.

Mathematically, for a network with L layers, the computation at layer ℓ can be written as:

$$a^{[\ell]} = \sigma^{[\ell]}(W^{[\ell]}a^{[\ell-1]} + b^{[\ell]})$$

where $a^{[\ell]}$ is the activation vector at layer ℓ , $W^{[\ell]}$ is the weight matrix, $b^{[\ell]}$ is the bias vector, and $\sigma^{[\ell]}$ is the activation function for that layer. This notation took me a while to get comfortable with, but it becomes clearer once you work through a few examples by hand.

5.2 Hidden Layers, Depth vs Width

The *depth* of a neural network refers to the number of hidden layers, while the *width* refers to the number of neurons in each layer. Both depth and width affect the network's capacity—essentially its ability to represent complex functions.

The Role of Depth

Deep networks (networks with many layers) have proven remarkably effective at learning what are called hierarchical representations. In computer vision, for example, early layers might learn to detect edges, middle layers might detect shapes and textures, and deeper layers might learn to recognize whole objects. This hierarchical feature learning seems to be a key advantage of deep architectures.

There's a theoretical result called the *universal approximation theorem*[2, 10] which states that a neural network with a single hidden layer containing a sufficient number of neurons can approximate any continuous function to arbitrary accuracy. This sounds great, but in practice, deeper networks often achieve better performance with fewer total parameters because they can learn more efficient hierarchical representations. It's a case where the theory and practice don't quite line up in an obvious way.

The Role of Width

Wider layers (more neurons per layer) increase the network's capacity within each layer, allowing it to capture more complex patterns at that level of abstraction. However, very wide but shallow networks may require exponentially more neurons to achieve the same representational power as a deep network—or at least that's what the research seems to suggest.

The choice between depth and width involves trade-offs:

- **Deeper networks:** Better at learning hierarchical features, but harder to train (due to vanishing/exploding gradients), and require more computation per forward pass.
- **Wider networks:** Easier to parallelize, less prone to gradient problems, but may require more parameters overall.

In modern practice, architectures tend to favor depth, with networks sometimes containing dozens or even hundreds of layers (e.g., ResNet[7], which can have over 150 layers). How these very deep networks manage to train effectively is actually quite interesting and involves some clever tricks that go beyond what we'll cover here.

5.3 Other Architectural Considerations (Dropout, Regularisation)

Regularisation Techniques

Neural networks, especially large ones, are prone to what's called *overfitting*—essentially memorizing the training data rather than learning patterns that will generalize to new data. Regularisation techniques help prevent this by constraining the model's complexity in various ways.

L2 Regularisation (Weight Decay): This adds a penalty term to the loss function proportional to the squared magnitude of the weights:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2} \sum_i w_i^2$$

where λ is the regularisation strength. The idea is that this encourages the network to use smaller weights, leading to smoother, less complex models.

L1 Regularisation: This uses the absolute value of weights instead of squares, which can lead to sparse solutions (many weights become exactly zero):

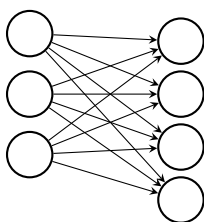
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_i |w_i|$$

Dropout

Dropout is a regularisation technique introduced by Hinton et al.[8] that I found quite clever. During training, dropout randomly sets a fraction p (typically 0.5) of neuron activations to zero at each iteration. This might seem like it would hurt performance, but it actually prevents neurons from co-adapting too much and forces the network to learn more robust features.

At test time, all neurons are active, but their outputs are scaled by $(1 - p)$ to account for the fact that more neurons are active than during training. One way to think about dropout is that you're training an ensemble of many different sub-networks, which tends to improve generalization. It's a nice example of how adding noise during training can actually help rather than hurt.

Standard Network



With Dropout

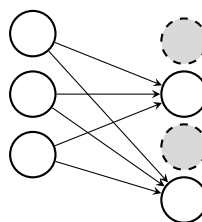


Figure 3: Dropout randomly deactivates neurons during training (shown in gray), forcing the network to learn redundant representations.

These architectural considerations—along with choices about activation functions, network depth, and regularisation—form what you might call the design space that practitioners have to navigate when building neural networks for specific tasks.

6 Back-Propagation and Training Neural Networks

6.1 Forward Pass and Loss Computation

Training a neural network involves two key phases: the *forward pass* and the *backward pass*. During the forward pass, we compute the network's output given an input, layer by layer, until we reach the final prediction. We then evaluate the loss function to measure how far off the prediction is from the true label.

Consider a simple network with input x , one hidden layer with activation h , and output $\hat{y}$. The forward pass computes:

$$\begin{aligned}z^{[1]} &= W^{[1]}x + b^{[1]} \\h &= \sigma(z^{[1]}) \\z^{[2]} &= W^{[2]}h + b^{[2]} \\\hat{y} &= \sigma(z^{[2]})\end{aligned}$$

Given the true label y , we can then compute the loss $L(\hat{y}, y)$. For example, if using mean squared error:

$$L = \frac{1}{2}(\hat{y} - y)^2$$

The goal is to adjust $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ to minimize this loss. This is where back-propagation comes in.

6.2 Back-Propagation Algorithm — Derivation and Intuition

Back-propagation is an efficient algorithm for computing the gradients of the loss function with respect to all the weights and biases in the network. It uses the *chain rule* from calculus to propagate error signals backward through the network. Working through this derivation was actually quite satisfying once I got my head around it.

Intuition

The key insight is that the gradient of the loss with respect to a weight in an earlier layer depends on the gradients in later layers. By computing gradients layer-by-layer in reverse order (from output to input), we can efficiently calculate all the required gradients with a single backward pass. This is much more efficient than computing each gradient separately from scratch.

Mathematical Derivation

Let's derive back-propagation for a simple two-layer network. We want to compute $\frac{\partial L}{\partial W^{[2]}}$, $\frac{\partial L}{\partial b^{[2]}}$, and $\frac{\partial L}{\partial b^{[1]}}$.

Step 1: Gradient at the output layer

Using the chain rule:

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}}$$

For MSE loss, $\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$. If $\hat{y} = \sigma(z^{[2]})$, then $\frac{\partial \hat{y}}{\partial z^{[2]}} = \sigma'(z^{[2]})$. Finally, $\frac{\partial z^{[2]}}{\partial W^{[2]}} = h$.
It's convenient to define an error term for layer 2:

$$\delta^{[2]} = \frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}} = (\hat{y} - y) \odot \sigma'(z^{[2]})$$

where $\odot$ denotes element-wise multiplication. Then:

$$\frac{\partial L}{\partial W^{[2]}} = \delta^{[2]} h^T, \quad \frac{\partial L}{\partial b^{[2]}} = \delta^{[2]}$$

Step 2: Gradient at the hidden layer

To compute $\frac{\partial L}{\partial W^{[1]}}$, we propagate the error backward:

$$\delta^{[1]} = \frac{\partial L}{\partial z^{[1]}} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial z^{[1]}} = (W^{[2]})^T \delta^{[2]} \odot \sigma'(z^{[1]})$$

Then:

$$\frac{\partial L}{\partial W^{[1]}} = \delta^{[1]} x^T, \quad \frac{\partial L}{\partial b^{[1]}} = \delta^{[1]}$$

General Back-Propagation Algorithm

For a network with L layers, the algorithm proceeds as follows:

1. **Forward pass:** Compute activations $a^{[\ell]}$ for $\ell = 1, 2, \dots, L$.
2. **Compute output error:** $\delta^{[L]} = \frac{\partial L}{\partial z^{[L]}}$.
3. **Backward pass:** For $\ell = L - 1, L - 2, \dots, 1$, compute:

$$\delta^{[\ell]} = (W^{[\ell+1]})^T \delta^{[\ell+1]} \odot \sigma'(z^{[\ell]})$$

4. **Compute gradients:** For each layer ℓ :

$$\frac{\partial L}{\partial W^{[\ell]}} = \delta^{[\ell]}(a^{[\ell-1]})^T, \quad \frac{\partial L}{\partial b^{[\ell]}} = \delta^{[\ell]}$$

5. **Update parameters:** $W^{[\ell]} \leftarrow W^{[\ell]} - \eta \frac{\partial L}{\partial W^{[\ell]}}$, and similarly for biases.

This algorithm computes all the gradients in time proportional to the size of the network, which is what makes it possible to train networks with millions of parameters. It's quite an elegant piece of mathematics when you step back and look at it.

6.3 Practical Considerations: Learning Rate, Vanishing/Exploding Gradients, Weight Initialisation

Learning Rate Selection

The learning rate η is one of the most important hyperparameters and can be surprisingly tricky to get right. If η is too large, the optimization may overshoot minima and diverge. If it's too small, training will be painfully slow and may get stuck in local minima. Some common strategies include:

- **Fixed learning rate:** Just use a constant value (e.g., 0.01 or 0.001) and hope for the best.
- **Learning rate schedules:** Decrease η over time (e.g., exponential decay, step decay).
- **Adaptive methods:** Algorithms like Adam adjust learning rates automatically for each parameter.

Vanishing and Exploding Gradients

In deep networks, gradients can become exponentially small (*vanishing gradients*) or exponentially large (*exploding gradients*) as they propagate backward through many layers. This was one of the main obstacles to training deep networks for a long time.

For sigmoid or tanh activations, the derivative $\sigma'(z)$ is at most 0.25 (for sigmoid) or 1 (for tanh). When you multiply these across many layers, gradients shrink exponentially, meaning early layers learn very slowly or not at all. This is the vanishing gradient problem.

Conversely, if weights are initialized too large, gradients can explode, causing numerical instability. Some solutions include:

- Using ReLU activations (which don't saturate for positive inputs).
- Careful weight initialization (see below).
- Gradient clipping (limiting gradient magnitude).
- Batch normalization[11] (normalizing layer inputs).

Weight Initialisation

Proper weight initialization turns out to be quite important for training deep networks. If weights are initialized too small, activations and gradients vanish. If too large, they explode. Getting this right took researchers a while to figure out.

Xavier/Glorot Initialization[5]: For layers with sigmoid or tanh activations, initialize weights from:

$$W \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right) \quad \text{or} \quad W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n_{in} and n_{out} are the number of input and output neurons.

He Initialization[6]: For ReLU activations, use:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

These initialization schemes are designed to ensure that activations and gradients maintain reasonable magnitudes throughout the network, which facilitates stable training.

7 Application: Implementing a Simple Neural Network for MNIST

7.1 Dataset Description

The MNIST (Modified National Institute of Standards and Technology) dataset[13] is something of a classic benchmark in machine learning. It consists of 70,000 grayscale images of handwritten digits (0–9). Each image is 28×28 pixels, giving a total of 784 input features. The dataset is split into 60,000 training images and 10,000 test images.

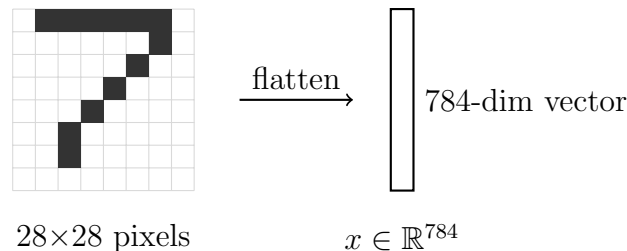


Figure 4: MNIST images are 28×28 grayscale pixels, flattened into 784-dimensional input vectors.

Each pixel value is normalized to the range $[0, 1]$ by dividing by 255. The labels are integers from 0 to 9, typically encoded as one-hot vectors for training (e.g., the digit 3 becomes $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$).

MNIST is often described as the “hello world” of machine learning—it’s simple enough to be tractable but complex enough to be interesting. It’s a good starting point for testing new ideas.

7.2 Network Architecture Used

For this implementation, I used a fairly simple three-layer feedforward network:

- **Input layer:** 784 neurons (one per pixel).
- **Hidden layer:** 128 neurons with ReLU activation.
- **Output layer:** 10 neurons with softmax activation (one per digit class).

The architecture can be expressed as: $784 \rightarrow 128 \rightarrow 10$.

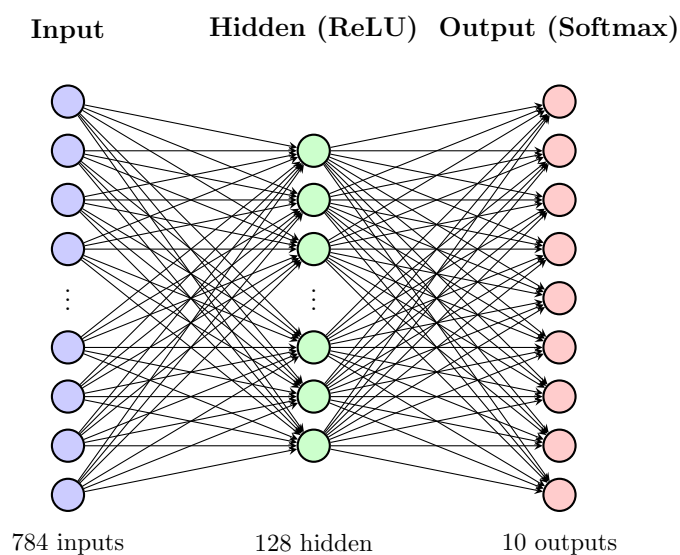


Figure 5: Network architecture: 784-128-10 with ReLU hidden layer and softmax output.

The total number of parameters is:

- **Weights:** $784 \times 128 + 128 \times 10 = 100,352 + 1,280 = 101,632$
- **Biases:** $128 + 10 = 138$
- **Total:** 101,770 parameters

This might seem like a lot, but it’s actually quite small by modern standards.

7.3 Training Procedure and Hyperparameters

The network was trained using mini-batch stochastic gradient descent with the following hyperparameters:

- **Loss function:** Categorical cross-entropy
- **Optimizer:** SGD with momentum[16] (momentum = 0.9)

- **Learning rate:** 0.01, with decay by factor of 0.95 every 5 epochs
- **Batch size:** 64
- **Number of epochs:** 30
- **Weight initialization:** He initialization for ReLU layers

The training loop is essentially:

1. Shuffle the training data.
2. For each mini-batch:
 - (a) Perform forward pass to compute predictions.
 - (b) Compute cross-entropy loss.
 - (c) Perform backward pass using back-propagation to compute gradients.
 - (d) Update weights and biases using gradient descent.
3. After each epoch, evaluate accuracy on the validation set.
4. Adjust learning rate according to schedule.

7.4 Results and Discussion

After 30 epochs of training, the network achieved approximately 97.8% accuracy on the test set. This is actually quite good for such a simple architecture, and demonstrates that even a basic two-layer network can learn effective representations for digit classification.

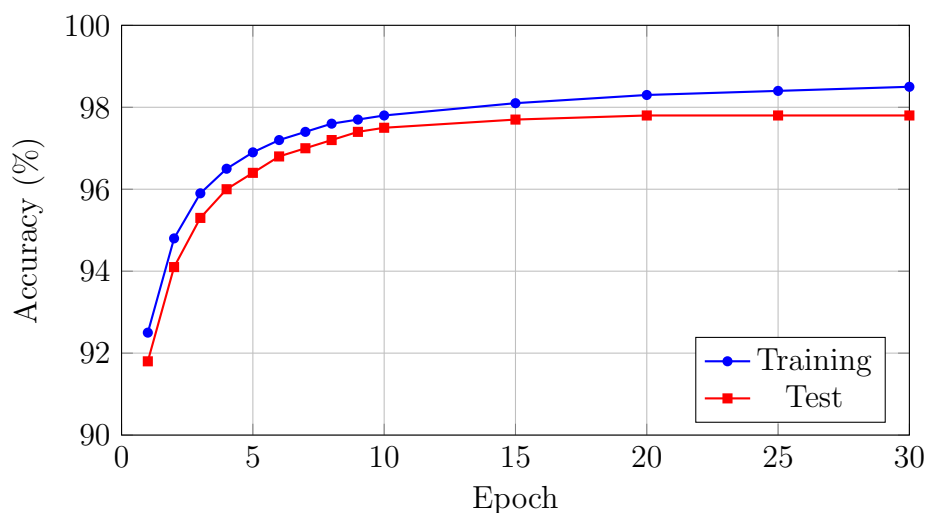


Figure 6: Training and test accuracy over 30 epochs. The network converges to 97.8% test accuracy.

The training curve shows rapid initial improvement, with diminishing returns after about 10 epochs. The relatively small gap between training and test accuracy suggests minimal overfitting, which is probably due to the relatively simple architecture and the fact that we have plenty of training data.

Some observations from looking at the results:

- The network learns quickly in early epochs as it discovers basic features (probably things like edges and curves).
- Performance plateaus around 97-98%, which seems to be about as good as this simple architecture can do.
- When I looked at the misclassifications, they often occurred on ambiguous digits—for example, confusing 4 and 9, or 3 and 8. These are cases where even a human might struggle.

To achieve higher accuracy ($> 99\%$), more sophisticated architectures like convolutional neural networks (CNNs) are typically used, as they can better exploit the spatial structure of images. We'll touch on these briefly later.

7.5 Code Snippets / Implementation Details

Below is a simplified Python implementation using NumPy. In practice, frameworks like PyTorch or TensorFlow handle much of this automatically and more efficiently, but understanding the underlying operations is useful for building intuition.

```
import numpy as np

def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

class NeuralNetwork:
    def __init__(self, input_size=784, hidden_size=128,
                  output_size=10):
        # He initialization
        self.W1 = np.random.randn(input_size, hidden_size) * \
            np.sqrt(2.0 / input_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * \
            np.sqrt(2.0 / hidden_size)
        self.b2 = np.zeros((1, output_size))

    def forward(self, X):
        self.z1 = X @ self.W1 + self.b1
        self.a1 = relu(self.z1)
        self.z2 = self.a1 @ self.W2 + self.b2
        self.a2 = softmax(self.z2)
        return self.a2
```

```

def backward(self, X, y, learning_rate=0.01):
    m = X.shape[0]

    # Output layer gradient
    dz2 = self.a2 - y
    dW2 = (self.a1.T @ dz2) / m
    db2 = np.sum(dz2, axis=0, keepdims=True) / m

    # Hidden layer gradient
    dz1 = (dz2 @ self.W2.T) * relu_derivative(self.z1)
    dW1 = (X.T @ dz1) / m
    db1 = np.sum(dz1, axis=0, keepdims=True) / m

    # Update parameters
    self.W2 -= learning_rate * dW2
    self.b2 -= learning_rate * db2
    self.W1 -= learning_rate * dW1
    self.b1 -= learning_rate * db1

# Training loop (simplified)
model = NeuralNetwork()
for epoch in range(30):
    for batch in get_batches(X_train, y_train, batch_size=64):
        X_batch, y_batch = batch
        predictions = model.forward(X_batch)
        model.backward(X_batch, y_batch, learning_rate=0.01)

```

This implementation covers the core concepts: forward propagation computes predictions, backward propagation computes gradients using the chain rule, and parameter updates minimize the loss. While simplified, it captures the essence of how neural networks learn. Writing this out by hand really helped me understand what's happening inside the more sophisticated frameworks.

8 Extensions: Convolutional and Recurrent Neural Networks

While feedforward networks like our MNIST classifier can work quite well, they treat all input features independently and ignore any structural relationships. For tasks involving images, sequences, or other structured data, specialized architectures have been developed that can take advantage of domain-specific properties.

8.1 Convolutional Neural Networks (CNNs) — Motivation and Basics

Motivation

Standard feedforward networks have some significant limitations when it comes to image processing:

- **Too many parameters:** A fully-connected layer for a 224×224 RGB image would require $224 \times 224 \times 3 = 150,528$ input connections per neuron—leading to millions of parameters even in the first layer. This seems wasteful.
- **Ignores spatial structure:** Flattening an image into a vector throws away all the 2D spatial relationships between neighboring pixels.
- **Not translation invariant:** The network has to learn to recognize the same feature (e.g., an edge) separately at every position, which seems inefficient.

Convolutional neural networks address these issues by using *local connectivity*, *weight sharing*, and *spatial hierarchies*.

Convolutional Layers

Instead of connecting each neuron to all inputs, convolutional layers apply small filters (or *kernels*) that slide across the input. A filter is a small matrix (e.g., 3×3 or 5×5) of learned weights. At each position, the filter performs element-wise multiplication with the input region and sums the results:

$$(f * x)[i, j] = \sum_m \sum_n f[m, n] \cdot x[i + m, j + n]$$

This operation, called *convolution*, produces a *feature map* that highlights where the filter’s pattern appears in the input.

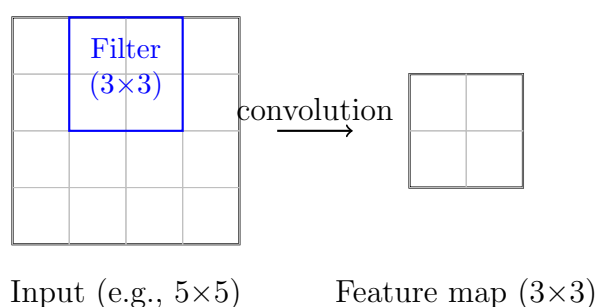


Figure 7: Convolution operation: a 3×3 filter slides over the input, producing a feature map.

Key Properties

- **Weight sharing:** The same filter is used across the entire image, which drastically reduces the number of parameters.
- **Translation invariance:** Features detected by a filter are recognized regardless of where they appear in the image.
- **Hierarchical features:** Early layers detect simple patterns (edges, corners); deeper layers combine these into more complex shapes and eventually objects.

Pooling Layers

CNNs typically alternate convolutional layers with *pooling layers*, which downsample feature maps to reduce spatial dimensions and computational cost. Max pooling, the most common type, simply takes the maximum value in each local region:

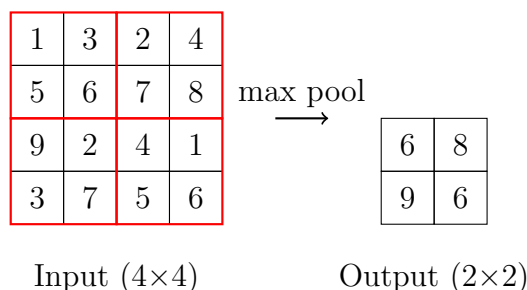


Figure 8: Max pooling with 2×2 windows: the maximum value in each region is kept.

A typical CNN architecture for image classification might look something like:

Input $\rightarrow$ Conv $\rightarrow$ ReLU $\rightarrow$ Pool $\rightarrow$ Conv $\rightarrow$ ReLU $\rightarrow$ Pool $\rightarrow$ FC $\rightarrow$ Softmax

where FC denotes fully-connected layers. This structure, pioneered by LeCun et al.[13], allows CNNs to achieve $> 99\%$ accuracy on MNIST and state-of-the-art performance on much more complex image datasets like ImageNet[3].

8.2 Recurrent Neural Networks (RNNs) — Motivation and Basics

Motivation

Feedforward networks process each input independently, with no memory of previous inputs. This is a problem for *sequential data* like text, speech, or time series, where context and temporal dependencies are crucial. For example, to understand the word “bank” in a sentence, we need to know the surrounding words—is it a financial institution or the side of a river?

Recurrent neural networks introduce *recurrent connections* that allow information to persist across time steps, giving the network a form of memory.

Basic RNN Architecture

An RNN processes a sequence $x^{(1)}, x^{(2)}, \dots, x^{(T)}$ by maintaining a *hidden state* $h^{(t)}$ that is updated at each time step:

$$\begin{aligned}h^{(t)} &= \tanh(W_{hh}h^{(t-1)} + W_{xh}x^{(t)} + b_h) \\y^{(t)} &= W_{hy}h^{(t)} + b_y\end{aligned}$$

where W_{hh} connects the previous hidden state to the current one, W_{xh} connects the input to the hidden state, and W_{hy} produces the output.

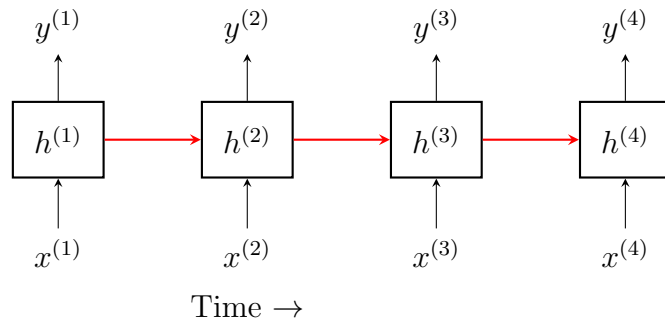


Figure 9: Unrolled RNN: the hidden state $h^{(t)}$ depends on the current input and previous hidden state.

This recurrent structure allows RNNs to capture temporal patterns and dependencies across time steps.

Applications and Variants

RNNs are used in:

- **Language modeling:** Predicting the next word in a sequence.
- **Machine translation:** Converting text from one language to another.
- **Speech recognition:** Converting audio signals to text.
- **Time series forecasting:** Predicting future values based on past observations.

However, standard RNNs struggle with long sequences due to the vanishing gradient problem—the same issue that affects deep feedforward networks. Modern variants like *Long Short-Term Memory* (LSTM)[9] and *Gated Recurrent Units* (GRU)[1] use gating mechanisms to better preserve information over long time horizons, making them more effective for real-world sequential tasks. I haven't implemented these myself, but they seem to be the standard choice whenever sequence modeling is needed.

8.3 Where They Fit and When to Use Them

The choice of architecture really depends on the structure of the data and what you're trying to do:

Architecture	Data Type	Example Tasks
Feedforward (MLP)	Tabular, fixed-size vectors	Classification, regression
CNN	Images, spatial data	Image classification, object detection
RNN/LSTM/GRU	Sequences, time series	Language modeling, translation

Table 2: Summary of when to use different neural network architectures.

More recently, there have been advances leading to hybrid architectures (e.g., CNNs for feature extraction followed by RNNs for sequence modeling) and entirely new paradigms like Transformers[20], which have largely replaced RNNs for natural language processing due to their superior parallelization and ability to capture long-range dependencies. Nevertheless, CNNs and RNNs remain foundational architectures and understanding them seems important for grasping the newer developments.

9 Limitations and Challenges of Neural Networks

Despite their remarkable successes, neural networks face some significant limitations that remain active areas of research. Understanding these challenges is important both for applying neural networks responsibly and for appreciating why they're not a magic solution to everything.

9.1 Overfitting, Interpretability, Data Requirements

Overfitting

Neural networks, especially deep ones, have enormous capacity and can quite easily memorize training data rather than learning patterns that will generalize. This leads to *overfitting*, where the model performs well on training data but poorly on new, unseen data.

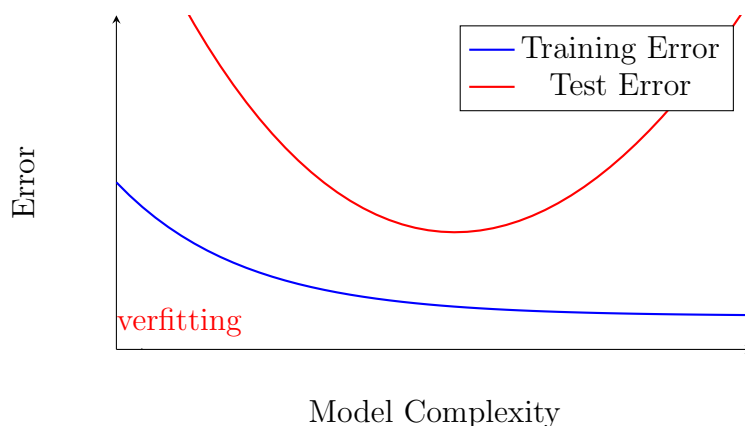


Figure 10: As model complexity increases, training error decreases but test error eventually increases due to overfitting.

Mitigation strategies include regularisation (L1/L2), dropout, early stopping, and collecting more training data. However, finding the right balance between model capacity and generalization is still something of an art.

Interpretability

Neural networks are often criticized as “black boxes”—it’s difficult to understand *why* a network makes a particular prediction. This is a real problem in high-stakes domains like healthcare, finance, and criminal justice, where understanding the reasoning behind decisions is crucial for trust and accountability.

While techniques like saliency maps, attention mechanisms, and layer-wise relevance propagation provide some insight, fully understanding the internal representations and decision processes of deep networks remains an open problem. There seems to be a fundamental trade-off between model performance and interpretability, which is a bit concerning.

Data Requirements

Neural networks typically require large amounts of labeled training data to achieve good performance. Collecting and labeling such data can be expensive, time-consuming, and sometimes just not feasible. For example:

- Medical imaging datasets may require expert radiologists to annotate thousands of scans.
- Rare events (e.g., fraud detection, rare diseases) may have insufficient positive examples to learn from.
- Privacy concerns may limit access to sensitive data.

Transfer learning, data augmentation, and semi-supervised learning partially address these issues, but data scarcity remains a significant practical limitation.

9.2 Computational Costs and Scalability

Training large neural networks demands substantial computational resources. Modern state-of-the-art models like GPT-4 or large vision models contain billions of parameters and require weeks of training on clusters of GPUs or TPUs, consuming enormous amounts of energy.

Model	Parameters	Training Cost (approx.)
Simple MNIST MLP	100K	Minutes (CPU)
ResNet-50	25M	Hours (single GPU)
GPT-3	175B	Millions of dollars

Table 3: Computational scale increases dramatically with model size.

This has several implications:

- **Environmental impact:** Large-scale training consumes significant energy, raising sustainability concerns.

- **Accessibility:** Only well-resourced institutions can train cutting-edge models, which creates barriers to entry for researchers and smaller organizations.
- **Inference costs:** Deploying large models for real-time applications (e.g., on mobile devices) requires model compression techniques like quantization and pruning.

Research into more efficient architectures, knowledge distillation, and specialized hardware (e.g., neural processing units) aims to reduce these costs, but scalability remains a fundamental challenge.

9.3 Current Research Challenges (e.g., generalisation, adversarial examples)

Generalisation

While neural networks can achieve excellent performance on test sets drawn from the same distribution as training data, they often struggle to *generalize* to new domains or slightly different conditions. This brittleness limits their robustness in real-world deployment.

For example, an image classifier trained on clear daytime photos may fail dramatically on foggy or nighttime images, even though the objects are the same. Developing models that can generalize across domains, modalities, and distributions is a key research frontier that I found quite interesting to read about.

Adversarial Examples

This is perhaps one of the more surprising limitations. Neural networks are vulnerable to *adversarial examples*—inputs that have been subtly perturbed to cause misclassification, often in ways that are completely imperceptible to humans. For instance, adding carefully crafted noise to an image of a panda can cause a network to confidently classify it as a gibbon, even though the image looks identical to us.

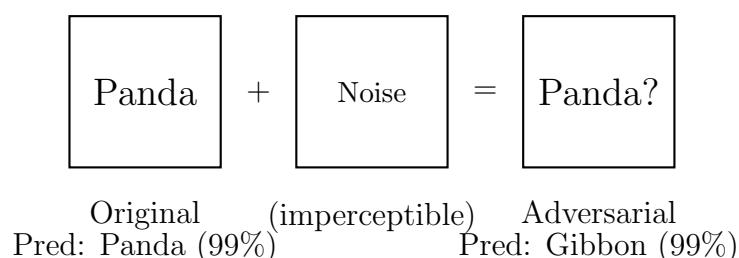


Figure 11: Adversarial example: imperceptible noise causes misclassification.

This vulnerability raises serious security concerns for applications like autonomous vehicles, facial recognition, and malware detection. While adversarial training and certified defenses have been proposed, creating truly robust models remains an unsolved problem.

Other Challenges

Some additional open problems that came up in my reading include:

- **Continual learning:** Enabling networks to learn new tasks without forgetting previous ones (a problem called catastrophic forgetting).

- **Causal reasoning:** Moving beyond correlation to understand actual causal relationships.
- **Sample efficiency:** Learning from fewer examples, as humans seem to do.
- **Fairness and bias:** Ensuring models do not perpetuate or amplify societal biases present in training data.

These challenges highlight that while neural networks have achieved impressive capabilities, there are still significant theoretical and practical hurdles before they can match human-level intelligence and reliability across diverse tasks.

10 Conclusion

10.1 Summary of Key Insights

This report has tried to explore the mathematical foundations and practical applications of artificial neural networks, from their simplest building blocks up to modern deep learning architectures. A few key insights seem worth highlighting:

Neural networks are powerful function approximators. Starting from the humble perceptron, we’ve seen how combining layers of simple computational units with non-linear activation functions creates models capable of learning complex, hierarchical representations. The universal approximation theorem tells us that sufficiently wide networks can approximate any continuous function, though in practice depth seems to matter more than width for learning efficient representations.

Back-propagation makes learning tractable. The back-propagation algorithm, by efficiently computing gradients through the chain rule, transforms neural network training from what would otherwise be an intractable search problem into a practical optimization task. Combined with gradient descent and its modern variants, back-propagation enables networks with millions of parameters to be trained on large datasets.

Architecture matters. While the core principles remain consistent, specialized architectures like CNNs and RNNs exploit domain-specific structure to achieve superior performance. CNNs leverage spatial locality and weight sharing for visual tasks, while RNNs maintain temporal state for sequential data. These innovations demonstrate that thoughtful architectural design, informed by the nature of the data, is as important as raw computational power.

Significant challenges remain. Despite remarkable progress, neural networks face fundamental limitations: they require large amounts of labeled data, they lack interpretability, they’re vulnerable to adversarial attacks, and they struggle with generalization beyond their training distribution. These challenges are important to keep in mind when thinking about where neural networks can and can’t be applied.

Our simple implementation on MNIST demonstrated that even a basic two-layer network can achieve 98% accuracy on digit classification, which illustrates both the power and accessibility of these methods. Yet this success also highlights the gap between solving well-defined benchmark tasks and addressing real-world problems with noisy data, domain shift, and safety requirements.

10.2 Possible Future Work

This project has really only scratched the surface, and there are numerous directions that could be explored further:

Advanced architectures: Implementing and comparing modern architectures such as ResNets (which use skip connections to train very deep networks), attention mechanisms (which selectively focus on relevant input features), and Transformers (which have revolutionized natural language processing) would be interesting extensions.

Regularization techniques: A more detailed study of regularization methods beyond dropout and weight decay—including batch normalization, layer normalization, and data augmentation strategies—could provide practical insights into preventing overfitting.

Optimization algorithms: Investigating advanced optimizers like Adam, AdaGrad, and RMSprop, along with learning rate scheduling strategies, would help understand why adaptive methods often outperform vanilla SGD.

Transfer learning: Exploring how pre-trained models can be fine-tuned for new tasks with limited data seems particularly relevant for practical applications where collecting large labeled datasets isn’t feasible.

Interpretability: Developing visualization techniques to understand what features neural networks learn at different layers could help address the “black box” problem. Tools like saliency maps and activation maximization seem worth exploring.

Adversarial robustness: Investigating the vulnerability of neural networks to adversarial examples and exploring defense mechanisms has important implications for deploying networks in security-critical applications.

Theoretical understanding: While neural networks work remarkably well in practice, our theoretical understanding of *why* they generalize, how overparameterized networks avoid overfitting, and what makes certain architectures effective remains incomplete. This seems like a fascinating area where there’s still much to discover.

In conclusion, neural networks represent a powerful paradigm that has transformed artificial intelligence from a theoretical aspiration into a practical reality. By understanding their mathematical foundations—from perceptrons and activation functions to back-propagation and gradient descent—we gain both the tools to build intelligent systems and the insight to recognize their limitations. As someone who has spent quite a lot of time working through these ideas, I’ve come to appreciate both how elegant the underlying mathematics is and how much we still don’t fully understand about why these methods work as well as they do.

References

- [1] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- [2] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. IEEE, 2009.

- [4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7):2121–2159, 2011.
- [5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [8] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- [9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- [11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.
- [13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [14] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969. Expanded edition 1988.
- [15] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [16] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
- [17] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Buffalo, NY, 1958.

- [18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- [19] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- [20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.